Documentation of the Eife-Algorithm Implementation for the Network Simulator (NS)

Morten Schlæger
TU-Berlin, TKN
July 2000

1. Abstract

This document describes the implementation of the TCP-Eifel algorithm[2] for ns[1]. It shows how Eifel is integrated into the existing ns object hierarchy and which files need to be modified. Further it explains how to set-up a simulation using the Eifel-Algorithm and presents some simulation results to visualize the operation of Eifel. Finally it shortly explains how to integrate the NS-Eifel distribution into an installed ns version.

This document is neither an introduction to ns nor to Eifel. A good place to start with the former is the ns homepage[1]. For the latter see [2].

2. Implementation of the Eifel-Algorithm

NS has different TCP classes which implement different features or versions of the TCP protocol. For example there are different classes for TCP/Thaoe and TCP/Reno. All of the TCP classes are derived from the TCP base class (see Figure 1).

In general there are two different possibilities to extend ns with a Eifel implementation. First, one could simply modify an existing TCP class to support Eifel. In this case compile-time or run-time flags could be used to enable or disable Eifel. The second approach is to build a own class for TCP-Eifel which is derived from one of the other TCP classes. Since the former approach does not fit well with the object oriented design a leads to ugly crowded source code we decided to follow the second approach.

Figure 1 shows a small part of the ns object hierarchy. This view is presented to show from which objects the Eifel object is derived.

TCPFull was chosen as the baseclass for the TCPEifel class. In opposite to the other TCP classes FullTCP does support connection management (exchange of SYN/FIN segments) and supports bidirectional data transfer. For the first version of the Eifel implementation it is important that the TCP timestamp option is supported. TCPFull implements this option.

2.1. Eifel interaction with TCP

The Eifel-Algorithm has four interaction points with TCP. These interaction points are summarized below:

1. Retransmission time-out: The retransmission timer handler must be extended. In case of the first time-out of a packet the values of snd_nxt, cwnd, and ssthresh must be saved. Further eifel_retransmit count must be initialized to one. In case of additional time-outs for the same packet eifel_retransmit must be incremented.

2. Dupack processing: When the third duplicate acknowledgment is received the corresponding Eifel code must be executed. In case of the first retransmission the same action as in case of the first time-out is required. In case of three duplicate acknowledgments after a
3. **Retransmission of a packet**: In case this is the first retransmission of a packet the corresponding timestamp is stored, otherwise no Eifel action is required.

4. **Acknowledgment processing**: When a new acknowledgment is received and we are in retransmission mode test whether retransmission was spurious. If this test evaluates to true Eifel actions are performed. The concrete action depends on how often the packet was retransmitted.

### 2.2. Interaction with base class

To allow the interactions with TCP as described above the TCPEifel class overloads some of the method of the TCPFull class. In case of a time-out a TCP object calls its `timeout_action` method. The `timeout_action` method of TCPEifel simply tests whether it is the first retransmission of the oldest unacked packet or not. If yes, the current congestion window and ssthresh as well as the time are stored in members. Otherwise the retransmission counter is incremented. Finally, this method calls the `timeout_action` method of its parent class (FullTCP). This method performs all the standard TCP stuff.

When the TCP objects receives three duplicate acknowledgment it calls the `dupack_action` method. The operation of this method is identical to the operation of the `timeout_action` method with two exceptions. In case that three duplicate acknowledgments are received after a time-out had already occurred, Eifel changes the TCP operation from slow start to congestion avoidance, Second the `dupack_action` of the baseclass is called instead of the `timeout_action` method.
When a new ACK is received TCP class calls the `ack_action` method. The `ack_action` method of TCPEifel tests for spurious retransmissions. If true it performs the Eifel actions otherwise it only resets the Eifel object members. Finally it calls the `ack_action` method of its baseclass to allow for standard ACK processing.

### 2.3. The TCPEifel Class

```cpp
class EifelFullTcpAgent : public FullTcpAgent {
public:
    EifelFullTcpAgent();

protected:
    double eifel_ts_first_rexmit_; // timestamp of first retransmission
    double eifel_last_cwnd_;      // value of cwnd before retransmission
    int eifel_last_ssthresh_;     // ssthresh value before retransmit
    int eifel_rexmit_seq_;        // seq of retransmitted segment
    int eifel_rexmit_no_;         // retransmission counter

// Eifel methods
    virtual void timeout_action();
    virtual void ack_action(Packet *);
    virtual void pack_action(Packet *);
    virtual void dupack_action();
};
```

Figure 2: Eifel class

### 3. Installation of Eifel

This Eifel version was implemented and tested using ns-2.1b6-current. The Eifel distribution for ns is a single tar file which was compressed using gzip. You can find this file under http://www-tkn.ee.tu-berlin.de.

The distribution contains a readme file and all the ns-eifel source files as well as some example scripts, which run some simple simulation, in order to show how to set-up a Eifel simulation. The tar file creates a directory ns-eifel. This directory contains the subdirs src, tcl and test. The src directory contains the c++ and header files. There you can find the source code of the eifel implementation as well as a additional module which implements a hiccup module. This module can be used to test the implementation of eifel and is used by my example scripts. It allows for simulating link outages and packet reordering without loosing packets.

The tcl directory contains tcl scripts which should be executed by `ns-lib.tcl` (see below). They are required to initialize the shared objects and to set some default values.

The test directory contains the example scripts. Just start ns with one of these scripts an enjoy to see eifel working.

To install Eifel perform the following steps:

1. Change to your ns installation directory and untar the Eifel.<version>.taz file.
2. Edit the makefile and add the following object files to the OBJ_CC variable
   eifel/src/tcp-eifel.o eifel/src/hiccup.o
3. Change to directory tcl/lib and edit the ns-lib.tcl file. Add the following lines just before ns-default is sourced.
   ```
   source ../../../eifel/tcl/ns-eifel.tcl
   source ../../../eifel/tcl/ns-hiccup.tcl
   ```
4. Add `set Agent/TCP/FullTcp/TCP_Eifel timestamp true` to your ns-default.tcl file
5. Go back to your ns top directory and run `make`

To start with your new eifel class change to the eifel/test directory and execute one of the sample scripts.

```
eifel
    src/
    ├── hiccup.cc
    │       Simulate outages and reordering
    │       hiccup.h
    │       tcp-eifel.cc
    │       tcp-eifel.h
    │
    │   tcl/
    │       ns-hiccup.tcl
    │       tcp-eifel.tcl
    │
    │   test:
    │       2nodes-hiccup.delay_once.tcl
    │       2nodes-hiccup.delay_rnd.tcl
    │       2nodes-hiccup.reorder_once.tcl
    │       2nodes-hiccup.reorder_rnd.tcl
    │       2nodes-hiccup.tcl
    │       extract.awk
    │       gen_cwnd_plot.awk
    │       gen_seq_plot.awk
    │       seq.plot.awk
    │
    │   plots
    │       plot.csh
    │       template.gnu
```

Figure 3: Contents of tar file

3.1. Usage of Eifel

The Eifel class can be used like any other agent in ns. Since it is derived form the TcpFull object all parameters of this object can be used as well. The Eifel object can be used as sender as well as receiver. However the Eifel algorithm itself will only be triggered when the Eifel object is the sender. If the Eifel object is connected to a TCPFull object it is required that the timestamp option of this object are enabled. The Eifel object enables this option by default.

4. The hiccup module

Hiccup provides functionality to simulate link outages and segment reordering without loosing a packet. It can be used for testing and playing with the Eifel class as well as for specific simulation scenarios.

A hiccup module was first used with the Eifel implementation and testing for BSD by R. Ludwig.

4.1. Implementation

The hiccup class is derived form the queue class and integrated into the ns link object (see Figure 4). It is located between the enqT object and the queue object. If there is no enqT object it
is placed at the top of the queue. This position between the enqueue trace object and the dequeue
trace object was chosen in order to have tracepoints when a packet is passed to the hiccup mod-
ule and when a packet is passed to the delay object (which simulates the link bandwidth and
propagation delay). Since no additional trace object is inserted between the hiccup object and
the queue object there are no tracepoints that show when a packet has left the hiccup module.
Although it should be possible to insert such a trace object it was not done since doing so would
require to change/add to the ns tracefile format.
The hiccup operates in either of four modes. In mode HICCUP_IDLE all packets are directly
passed to the next link object. In HICCUP_DELAY mode packets are queued until the mode is
changed back to HICCUP_IDLE. Then all packets are passed to the neighbor object in a single
no time consuming burst. In mode HICCUP_RESORT the queueing of a single pack et is de-
layed until resort length later pack ets are queued. The last mode HICCUP_CONG allows to
drop packets. Dropped packets are passed to the drop tracer if one is connected. This mode is
only finished when hiccup is set to a new mode. Currently it is not possible to put the hiccup
mode in two modes simultaneously e.g. reorder and delay packets. However this functionality
can be added easily where it makes sense.

4.2. Usage
Since the hiccup object is not a standard part of the ns link it must be inserted explicitly into the
link. This operation is identical to adding a error generator to the link (see the ns manual). First
of all a new hiccup module must be created using the tcl command new. To insert this new hic-
cup module into a link between two nodes a new ns method \texttt{insert-hiccup} was written. This
method takes as arguments the source and destination node and the hiccup object. The mode of
the hiccup object can be set by calling one of the hiccup method as described below. Figure 5
shows an example code fragment. Figure 6 list the different methods available to tcl.

![Figure 4: NS Link object](image)

**set h [new Queue/Hiccup]**

\texttt{\$ns insert-hiccup $n0 $n1 $h}

\texttt{\$ns at 0.10 "$h setDelay"}

\texttt{\$ns at 0.5 "$h setIdle"}

**Figure 5: Example tcl-code fragment**
Please note that ns only knows about simplex links. Therefore only packets send in one direction are affected by hiccups. If link outages, etc. should be simulated in both directions a hiccups module must be inserted in both links.

5. Some Eifel test simulation

The purpose of the following simulations is to show the operation of Eifel for the different triggering events like link outages and reordering. The following table summarizes the different scenarios and gives an overview about how the Eifel algorithm will modify the TCP behavior in case it detects a spurious retransmission. In this table we assume the that Congestion Window, Slow Start threshold and the next packet not send so far had the following values when either a timeout had occurred or when the third duplicate acknowledgment was received: $cwnd$, $ssthresh$, $snd_nxt$.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Cong.Wnd</th>
<th>Slow Start threshold</th>
<th>Next Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spurious Timeouts</td>
<td>1</td>
<td>$cwnd$</td>
<td>$ssthresh$</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>max($cwnd/2,1$)</td>
<td>max($cwnd/2,2$)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Fast Retransmit</td>
<td></td>
<td>$cwnd$</td>
<td>$ssthresh$</td>
</tr>
</tbody>
</table>

Table 1: Effects of Eifel on TCP behavior
5.1. Set-up

The simulation set-up is composed from two stations of which one operates as the sender and the other one as the receiver. The stations are connected by a 1Mbit/s duplex link with a one way latency of 100ms\(^1\). The basic protocol stack on both stations consist of the TCP and IP protocol. The TCP version on the sender implements the Eifel algorithm according to the BSD patch 1.1. On both TCP entities the timestamp option is enabled. The TCP-Eifel implementation is derived from ns’s full tcp class.

In order to allow controlled link outages and packet reordering the standard ns link was extended by a ‘hiccup’ module. The hiccup module was only used for the data transmission. The acknowledgment were not affected by its operation.

For tracing the standard ns trace points were used (shown shaded in the figure). Each time TCP transmits a packet a tracepoint is recorded (enqueue event). Each time a packet is removed from the queue for transmission a tracepoint is recorded (dequeue event). If a packet is dropped because of a queue overflow or the hiccup mode a drop event is recorded. Finally when a packet has reached the destination station a tracepoint is recorded (receive event). The congestion window is recorded by the TCP process. In the following figures we will only use the enqueue event for outgoing packets at the sender and the receive event for incoming acknowledgments also at the sender. These tracepoints correspond to the typical TCP sequence and acknowledgment number plots with which most readers are supposed to be familiar.

5.2. Scenario 1 - Basic Simulation

In scenario 1 the hiccup module was idle during the whole run. Since the queue length of the link was also unlimited no packet drop could happen. Thus there were no events which could have triggered the Eifel-Algorithm. This test were performed to show that the Eifel Implementation does not influence TCP behavior in such a case.

\(^1\) For the test simulations these values were chosen arbitrary.
To show this the simulation was run with Eifel enabled and disabled. The traces of both runs were compared using the unix tool ‘diff’. The comparison shows no differences. No curves are shown for this scenario.

5.3. Scenario 2 - Congestion Losses

Again the purpose of this scenario is to show that the Eifel implementation for ns does not affect normal TCP operation as long as there are no spurious retransmissions. As in the first scenario simulations runs with and without Eifel enabled were run and compared. However this time the simulation script has put the hiccup module into the ‘Congestion’ mode at a specific point in time and back to the ‘Idle’ mode later which means that some packets were dropped. Again the comparison of the tracefiles showed no difference. Again, no curves are shown.

5.4. Scenario 3 - Outages during slow start

In this scenario the hiccup module was programmed to introduce a link outage long enough that a single packet was retransmitted by TCP either once, or twice, or three times. However, as mentioned above in the description of the hiccup module no packet were lost. Thus, with this test we checked the operation of our Eifel implementation in case of spurious time-outs. The outage period of the next three scenarios happened during TCP’ slow start phase.
5.5. Scenario 3a - Single Retransmission

Figure 8 shows curves representing the transmission of segments and receptions of corresponding acknowledgment by the sending machine. For this purpose the events ‘enqueue packet’ on the data path and the event ‘received packet’ on the acknowledgment path were used. As shown in the figure a single packet is retransmitted about time-point 1.1ms. In spite of this retransmission the sender continues with normal operation (does not perform GbN) after the corresponding acknowledgment is received, indicating that the Eifel implementation works proper in this case.

Figure 10 shows the development of the congestion window. At time-point 1.1ms when the packet is retransmitted the congestion window is reduced to one. However, instead of performing the slow start algorithm starting from 1 the congestion window is restored and normal operation continues after the missing acknowledgment is received about time-point 1.2ms.
5.6. Scenario 3b - Two retransmissions

Same scenario as above but this time the outages period was increased so that two time-outs for same packet occurred at the sender leading to two retransmissions of the same packet. Figures 11-14 show the corresponding plots. Figure 11 shows two retransmissions of the same segment. The first one about segment 1.5 ms and the second one about 2.6 ms. After the acknowledgment has shown that the time-outs were spurious TCP continues with transmitting the next packet which was not sent before. In figure 13 one can observe that this time the congestion window...
is not restored to its old value but to the half of the old value. Further the ssthresh hold is also set to cwnd/2 leading to congestion avoidance.

![Graph showing TCP-Eifel and two retransmissions of a single packet.](image1)

Figure 11: TCP-Eifel and two retransmissions of a single packet

![Graph showing an interesting part of figure 11 in more detail.](image2)

Figure 12: Interesting part of figure 11 in more detail
5.7. Scenario 3c - Three retransmissions

In this scenario the outage phase is again increased so that three spurious time-outs have occurred for a single packet leading to three retransmissions of a single packet. Figure 15 shows that Eifel prevents the GbN behavior of TCP after it has detected that the time-outs were spuri-
ous. However this time Eifel leaves cwnd and ssthresh untouched since more than two time-outs had occurred. This is depicted in figure 17.

Figure 15: Three timeouts after a long link outage

Figure 16: More detailed view on figure 15
5.8. Scenario 4 - Reordering

In this scenario hiccup was implemented to reorder packets, so that a fast retransmit has occurred.

Since the packet were only reordered and no packet was lost the retransmission due to the three duplicate acknowledgments is spurious. The operation of TCP is depicted in figure 18 and in figure 20. Figure 21 extracts the congestion window from figure 20. As can be seen from this figure the congestion window is restored to its old value after Eifel has detected that the retransmission was spurious.
Figure 19: More details from figure 18

Figure 20: Showing the congestion window together with figure 19
5.9. Three Duplicate Acknowledgments after a timeout

In this scenario a real segment loss has occurred. Hiccup was configured to drop a single packet a delay all following packets without losing an additional packet. Due to the sudden increase in round trip time the TCP timer expires before any duplicate acknowledgment is received. Some time later the duplicate acknowledgment arrive. After the third duplicate acknowledgment the Eifel algorithm puts TCP into congestion avoidance /fast recovery mode without repeating the missing packet again.

6. References


Figure 21: Development of congestion window